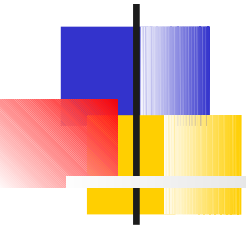
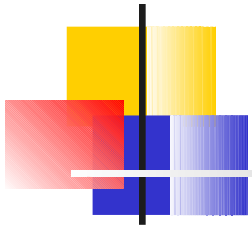
Welcome



链 表

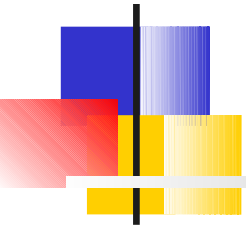
北京亚嵌教育研究中心



本次课程内容大纲

- 什么是链表？
- 链表
  - 构造链表 - 动态内存申请
  - 单向链表
  - 双向链表

Section 1



什么是链表

链表

什么是链表：

- 链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

链表

- 链表与数组的优缺点：

1、链表的大小可以动态变化，数组的长度不可动态变化。

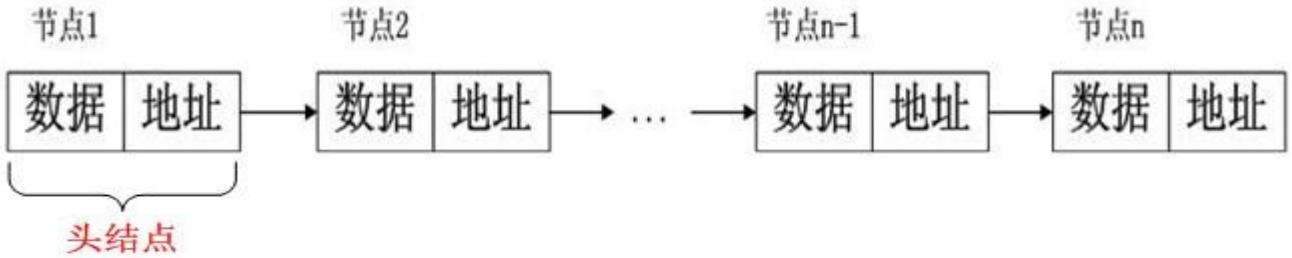
2、数组是线性存储，数据可以进行随机存取；链表是链式存储，不可随机存取，只能顺序存取。

3、数组插入或删除元素时，需要大规模搬移数据；链表插入或删除数据比较方便，不需要进行搬移，只进行指针的修改。

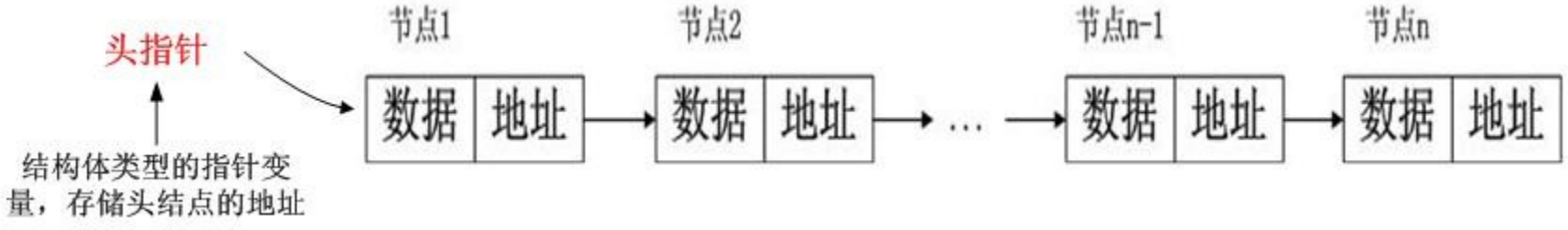
链表

链表的两个术语：

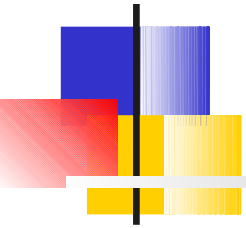
1、头结点：链表的首个数据



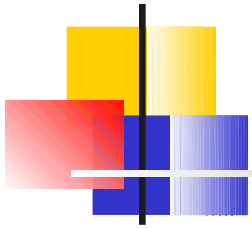
2、头指针：指向链表的头结点



Section 2



链表使用

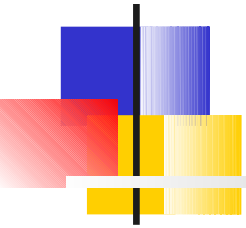


链表使用

- 构建链表 - 动态内存申请
- 单向链表
- 双向链表

链表使用

- 链表共有特性：
  - 基本的存储方式不变：每个结点物理存取区间不连续
  - 链表的每个结点中必然至少有一个指针数据，用于和其他结点进行链接
  - 链表结点为结构体类型



构建链表 - 动态内存申请

构建链表 - 动态内存申请

- 堆空间分配函数
- **void *malloc(size_t size)**
  - 功能：动态申请内存
  - 参数解释： **size** 代表要申请的内存字节数
  - 返回值：成功：返回从堆区申请到的 **size** 个 字节的内存的首地址

    失败：返回 **NULL**，申请空间失败

构建链表 - 动态内存申请

- 堆空间释放函数
- **void free(void *ptr);**
  - 参数解释： **ptr** 只能是动态申请到的内存块的首地址
  - 功能：释放动态申请的内存
  - **free** 应该和 **malloc** 等函数成对出现

构建链表 - 动态内存申请

- 堆空间释放函数
- **void free(void *ptr);**
  - **free 函数出错原因：**
    - 释放的不是动态申请的内存
    - **ptr** 不代表动态内存的首地址

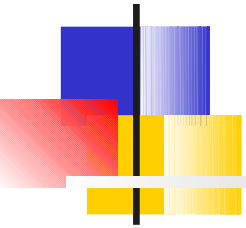
构建链表 - 动态内存申请

- 堆空间释放函数
- **void free(void *ptr);**
  - 保证了资源的及时清理
  - 释放内存后，原来申请的内存空间不能再用， **ptr** 被称作野指针，不可再有 ***ptr = …** 的操作，通常在释放后会将 **ptr = NULL**

构建链表 - 动态内存申请

- 堆空间申请、释放函数使用基本步骤
  - **Step1： p = malloc(size);**
  - **Step2： 使用申请到的内存**
  - **Step3： free(p);**

    p 的值在第 2 步中不可被修改
  - **Step4： p = NULL;**



单向链表

单向链表

单向链表：

- 类似于上图结构，每个链表结点都有唯一一个指向下一个结点的指针用于进行结点间的相连，构成如上形式的链表称为单（向）链表。
- 既然如此，那么结点数据的类型如何定义？

单向链表

单向链表：

- 结点类型定义：

```
typedef struct node{
    data1;
    data2;
    …
    datan;
    struct node *next; // 结点的相连
}node_t;
```

单向链表

- 单向链表基本操作：
  - 创建
  - 操作：查找、删除、插入
  - 销毁

单向链表

- 单向链表基本操作：

```
typedef struct node{
    int data;
    struct node *next;
}node_t;
 node_t *head = NULL;
```

单向链表

- 单向链表基本操作：

1、创建：链表中结点个数（链表占用空间的大小）是根据应用需要动态变化的，因此进行链表创建时需要使用动态空间申请函数实现。

链表创建基本步骤：

1、申请空间

2、向空间中存放数据

3、插入链表：头部链入或尾部链入

单向链表

- 单向链表基本操作：

2、操作：

插入结点： 1、在链表头插入

2、在链表尾部插入

3、在指定位置插入

单向链表

- 单向链表基本操作：

2、操作：

在链表尾部插入结点：

实现 **node_t *link_insert(node_t *head, int val)** 函数

参数： **node_t *head**： 链表头指针

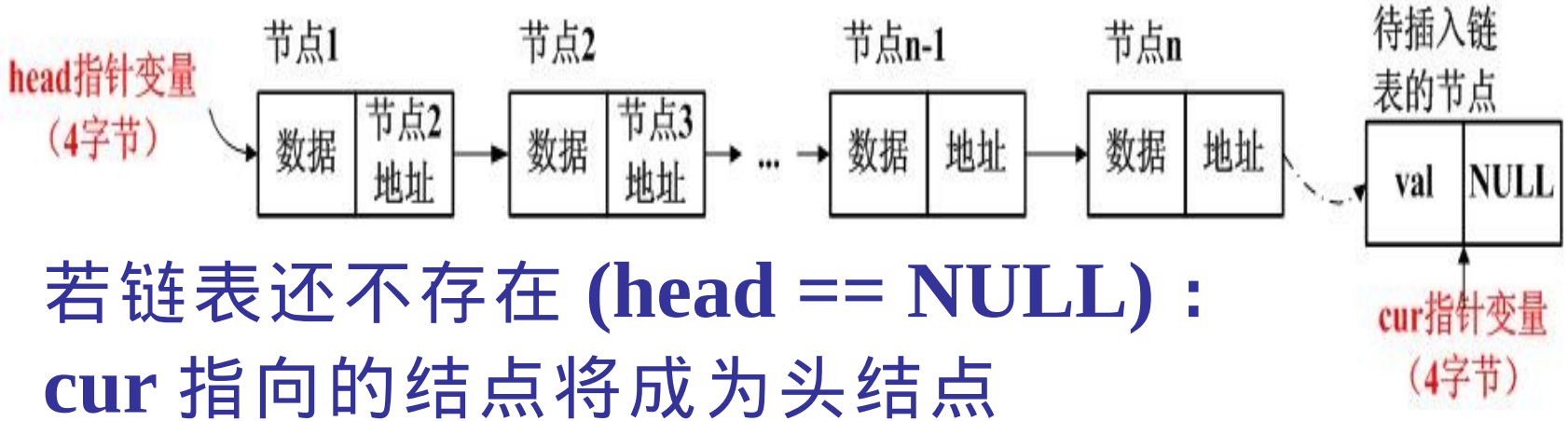
int val： 待插入的值

返回值：添加了新结点的链表的头指针

单向链表

2、操作：

在链表尾部插入结点：



若链表还不存在 **(head == NULL)**：**cur** 指向的结点将成为头结点

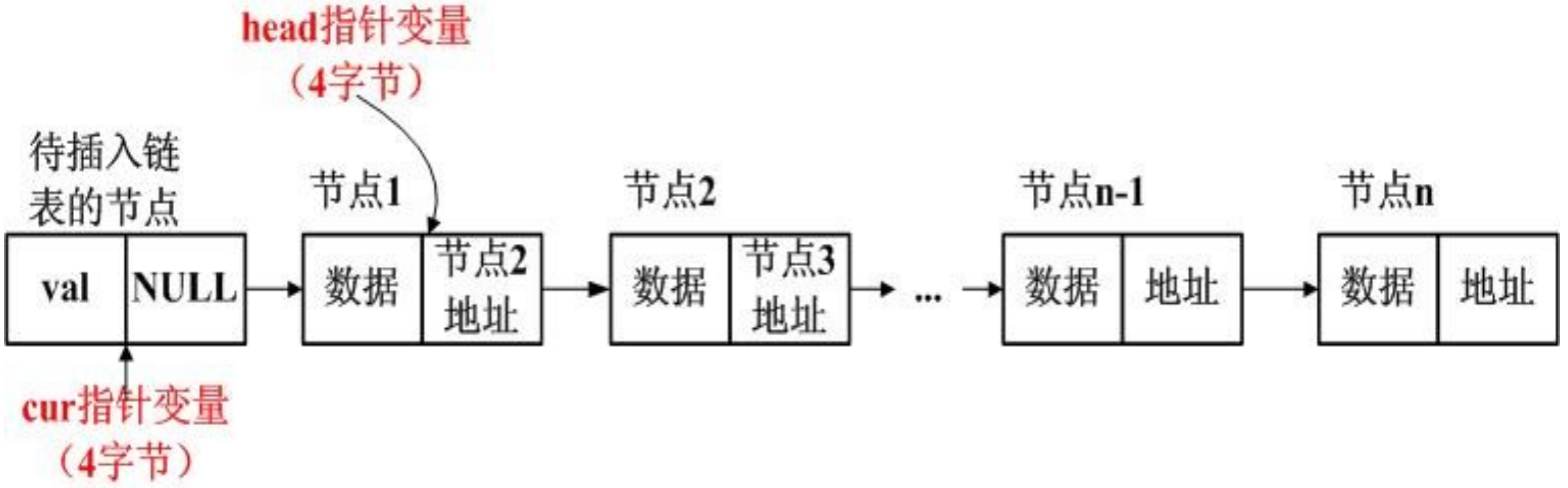
若链表已经存在：要从头结点开始顺序向后查找一直到最后一个结点，**cur** 指向的结点链接在最后一个结点的后面

单向链表

2、操作：

在链表头部插入结点：

编写程序，完成从链表头部进行结点插入。



单向链表

- 单向链表基本操作：

2、操作：

在指定位置插入结点：与链表创建比较类似

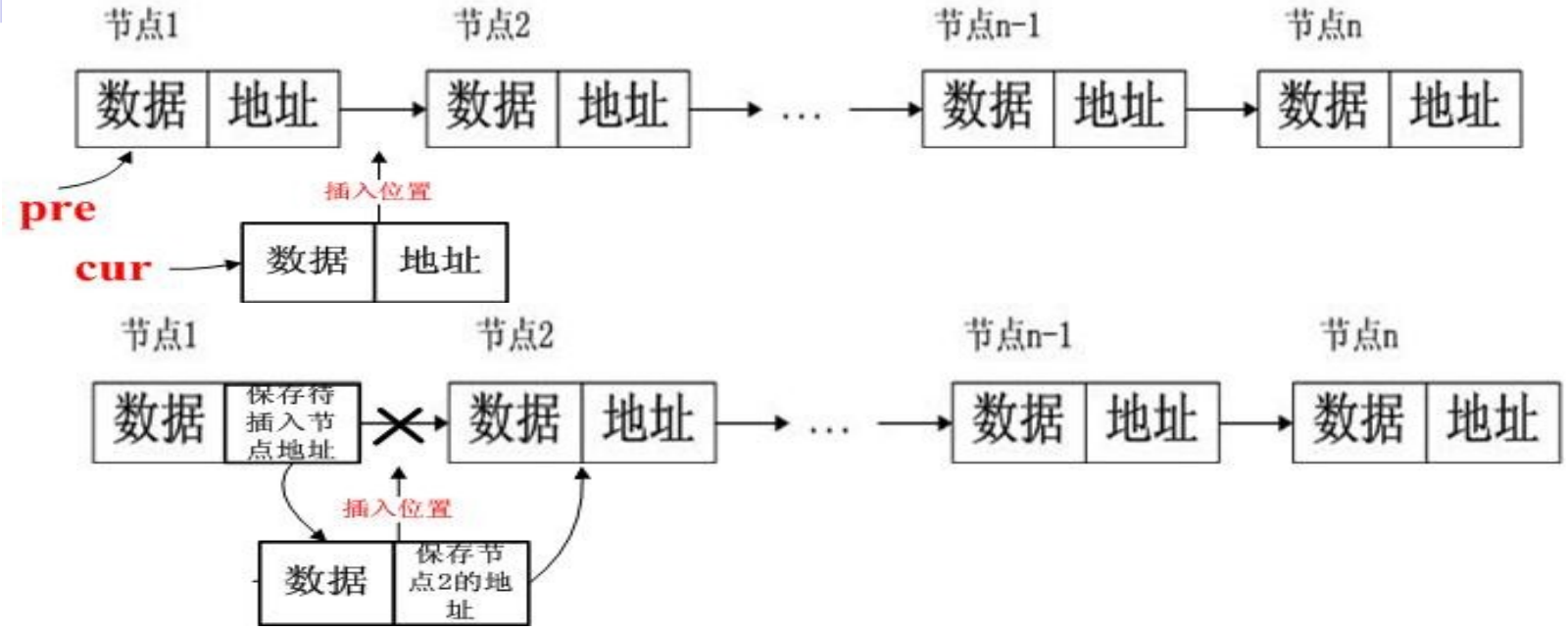
a、构造将要插入到链表中的结点用 cur 指针指向

b、找到该结点待插入位置前一结点的指针 pre

c、修改指针值，插入数据：

cur-> next = pre->next

单向链表



cur-> next = pre->next

pre->next = cur;

单向链表

- 单向链表基本操作：

2、操作：

删除结点：从链表中摘除一个结点

a、找到待删除结点 cur 及其前导结点 pre

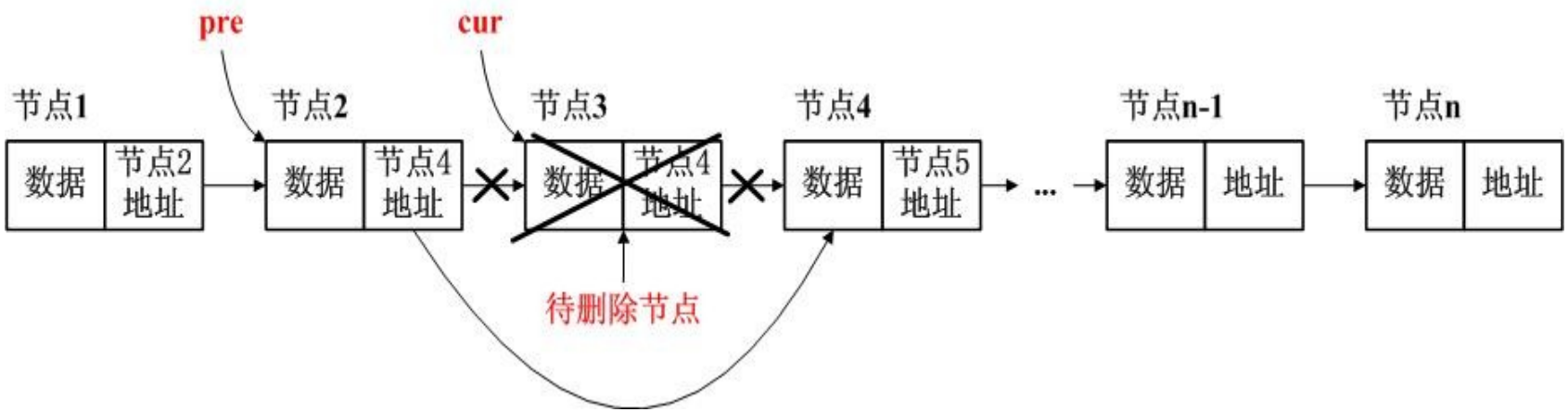
b、修改结点的指针成员

```
del = cur
pre->next = cur = cur->next
```

c、释放空间： free(del)

单向链表

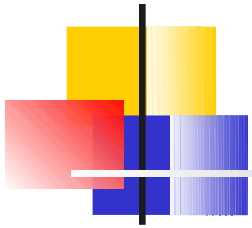
删除结点：



del = cur ;

pre->next = cur = cur->next ;

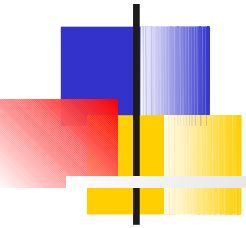
free(del) ;



单向链表

- 单向链表基本操作：

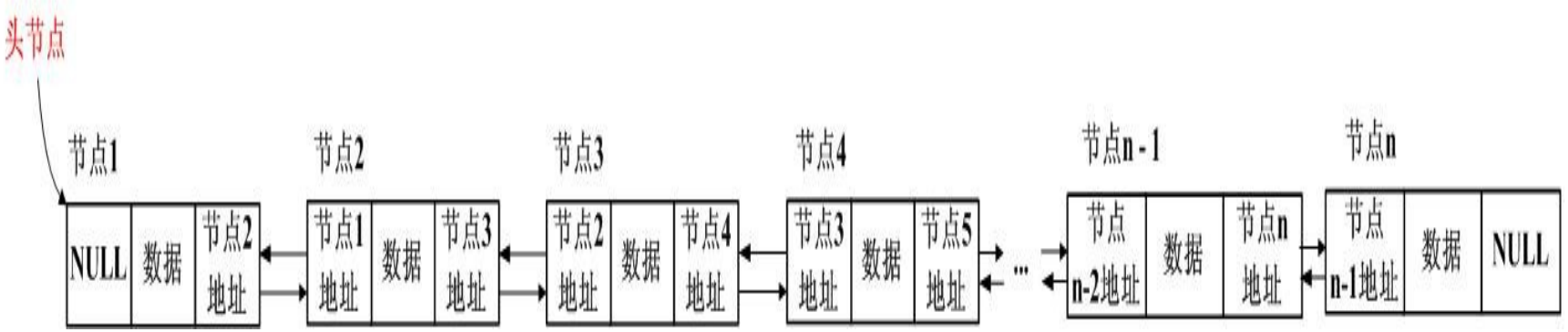
  3、销毁链表：删除链表中每个结点



双向链表

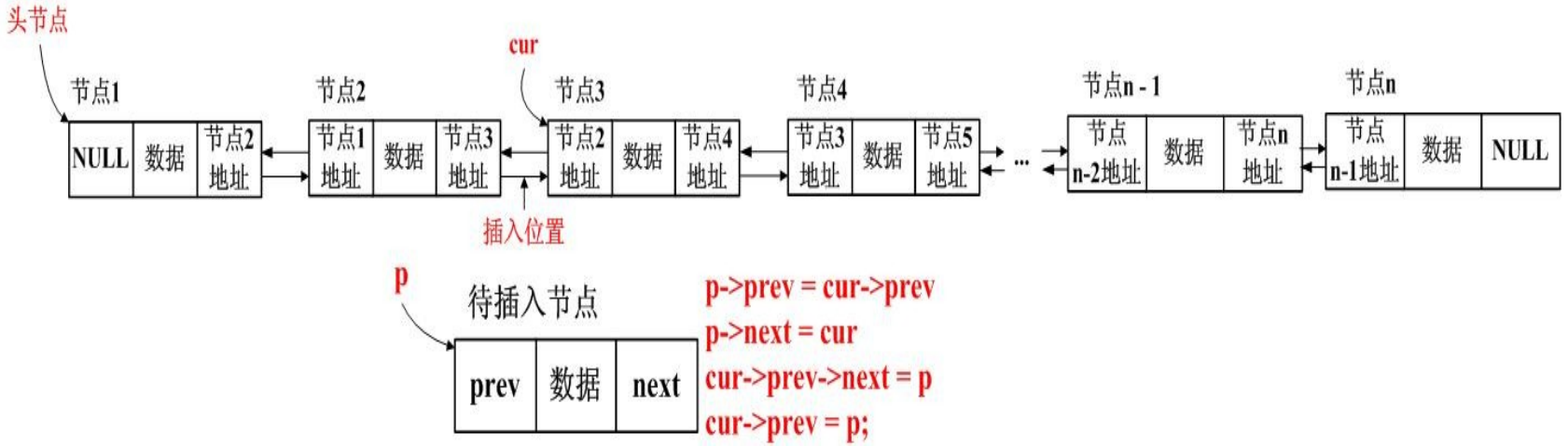
双向链表

- 双向链表：
  - 链表结点的成员中除了有指向下一个结点的 **next** 指针，又增加一个指向前导结点的指针 **prev**。



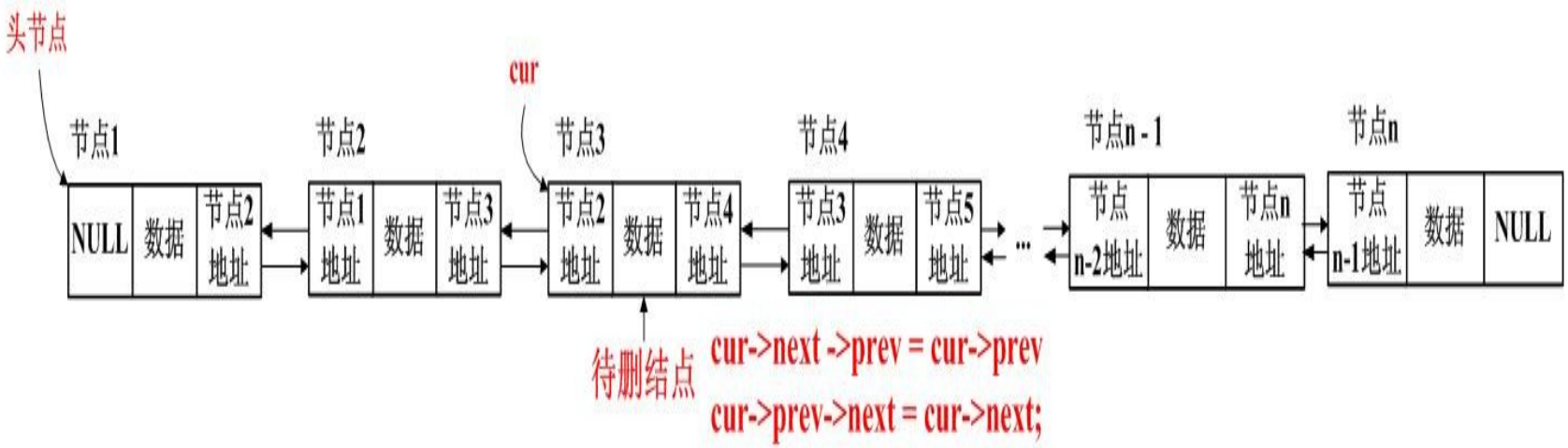
双向链表

- 双向链表操作：
  - 插入、删除：双向链表多了一个 **prev** 指针，使得插入、删除结点的操作变得更



双向链表

- 双向链表操作：
  - 插入、删除：双向链表多了一个 **prev** 指针，使得插入、删除结点的操作变得更



双向链表

- 双向链表操作：
  - 创建：相对于单向链表，双向链表的每个结点的 **next**、 **prev** 指针需要指向后续及前导结点。
  - 插入、删除：双向链表多了一个 **prev** 指针，使得插入、删除结点的操作变得更简单。

链表

练习：

1、 创建一个有 5 个结点的单链表

```
struct student{
    unsigned id;
    char name[20];
    struct student *next;
};
```

2、根据上题创建的链表，删除指定 name 的结点。

链表

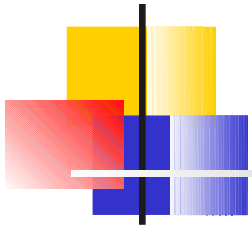
练习：

3、链表实现约瑟夫环问题。

4、实现某个单链表的逆序。

5、如何判断一个链表中是否存在环。

6、使用一个链表来表示一个任意长度的超长正整数，然后实现两个正整数（等长）的加法。



Let's DO it!

Thanks for listening!